CMSC 330: Organization of Programming Languages

Type Inference and Unification

Type Checking vs Type Inference

- Type checking: use declared types to check types are correct

```
let apply (f:('a->'b)) (x:'a):'b = f x
```

- Type inference:

```
let apply f x = f x
```

  - Infer the most general types that could have been declared, and type checks the code without the type information

The Type Inference Algorithm

- Input: A program without types

- Output: A program with type for every expression, which is annotated with its most general type

Why do we want to infer types?

- Reduces syntactic overhead of expressive types
  - // C++ Declare a vector of vectors of integers
    std::vector<std::vector<int>> matrix;
- Guaranteed to produce most general type
- Widely regarded as important language innovation
- Illustrative example of a flow-insensitive static analysis algorithm

History

- Original type inference algorithm
  - Invented by Haskell Curry and Robert Feys for the simply typed lambda calculus in 1958
- In 1969, Hindley
  - extended the algorithm to a richer language and proved it always produced the most general type
- In 1978, Milner
  - independently developed equivalent algorithm, called algorithm W, during his work designing ML
- In 1982, Damas proved the algorithm was complete.
  - Currently used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0. Have been plans for Fortress, Perl 6, C++0x,…

Type Inference: Basic Idea

- Example

```
fun x -> 2 + x
-: int -> int = <fun>
```

- What is the type of the expression?
  - + has type: int → int → int
  - 2 has type: int
  - Since we are applying + to x we need x : int
  - Therefore, **`fun x -> 2 + x`** has type int → int

Type Inference: Basic Idea

- Example

```
fun f  =>  f 3
  -:(int → a) →  a = <fun>
```

- What is the type of the expression?
  - 3 has type: int
  - Since we are applying f to 3 we need f : int → a and the result is of type a
  - Therefore, **fun f → f 3** has type (int → a) →a

Type Inference: Basic Idea

- Example

```
fun f → f (f 3)
```

- What is the type of the expression?

Type Inference: Basic Idea

- Example

```
fun f → f (f 3)
```

- What is the type of the expression?

```
(int -> int) -> int
```

Type Inference: Basic Idea

- Example

```
fun f → f (f "hi")
```

- What is the type of the expression?

Type Inference: Basic Idea

- Example

```
fun f → f (f "hi")
```

- What is the type of the expression?

```
(string -> string) -> string
```

Type Inference: Basic Idea

- Example

```
fun f → f (f 3, f 4)
```

- What is the type of the expression?

Type Inference: Basic Idea

- Example

```
fun f → f (f 3, f 4)
```

- What is the type of the expression?

```
Type error!
```

Type Inference: Example

```
let square = fun z → z * z in
fun f → fun x → fun y →
  if (f x y) then (f (square x) y)
  else (f x (f x y))
```

Type Inference: Example

```
let square = fun z → z * z in
fun f → fun x → fun y →
  if (f x y) then (f (square x) y)
  else (f x (f x y))
```

```
*: int → (int → int)
      z : int
      square : int → int
```

Type Inference: Example

```
let square = fun z → z * z in
fun f → fun x → fun y →
  if (f x y) then (f (square x) y)
  else (f x (f x y))
```

```
* : int → (int → int) → z : int → square : int → int
```

```
f : 'a → ('b → bool), x: 'a, y: 'b
```

Type Inference: Example

```
let square = fun z → z * z in
fun f → fun x → fun y →
  if (f x y) then (f (square x) y)
  else (f x (f x y))
```

```
* : int → (int → int) → z : int → square : int → int
```

```
f : 'a → ('b → bool), x: 'a, y: 'b
```

```
a: int
```

Type Inference: Example

```
let square = fun z → z * z in
fun f → fun x → fun y →
  if (f x y) then (f (square x) y)
  else (f x (f x y))
```

`* : int → (int → int) → z : int → square : int → int`

`f : 'a → 'b → bool, x: 'a, y: 'b`

`a: int`

`b: bool, y is the second argument of f. y and (f x y) have the same type. (f x y): bool`

Type Inference: Example

```
let square = fun z → z * z in
fun f → fun x → fun y →
  if (f x y) then (f (square x) y)
  else (f x (f x y))
```

```
* : int → (int → int) → z : int → square : int → int
```

```
f : 'a → 'b → bool, x: 'a, y: 'b
a: int
b: bool
```

```
(int → bool → bool) →int →bool → bool
```

Unification

- Unification is an algorithmic process of solving equations between symbolic expressions
- Unifies two terms
- Used for pattern matching and type inference
- Simple examples
  - int * x and y * (bool * bool) are **unifiable**
    - y = int
    - x = (bool * bool)
  - int * int and int * bool are **not unifiable**

Type Inference Algorithm

- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: literals (2), built-in operators (+), known functions (tail)
  - From form of parse tree: e.g., application and abstraction nodes
- Solve constraints using *unification*
- Determine types of top-level declarations

Type Inference: Example

```
fun x -> 2 + x

       int -> int -> int
```

```
Type: Guess -> type of
2 + x:Guess

int -> int
```

Type Inference: Function Application

```
(fun x-> fun y -> x=y) 1;;

App (e1, e2) ->
   let t1 = infer e1 env in
   let t2 = infer e2 env in
   let g = fresh_guess () in
   unify t1 (TArrow (t2, g));
g
```

(‘a->(‘a->Bool)

int

(‘a->(‘a->Bool) ==
(int -> Guess)

‘a=int
Guess= (‘a->Bool) =(int -> bool)

Inferring Polymorphic Types

Unconstrained type variables become polymorphic types

```
Fun x-> x:  Guess -> Guess

'a -> 'a
```

Recognizing Type Errors

```
let x = 10 in x=true
```

```
TypeError "unify failure: TInt <> TBool".
```

Let Polymorphism

- Let polymorphism is formalized in the **Hindley–Milner** type system:
  - **Generalization:** When a value is bound to a name using let, the type variables that don't appear in the environment are generalized to be universally quantified.
  - **Instantiation:** Each use of that variable can have its generalized type instantiated to a concrete type.

```
let id x = x in (id 1, id true)
```

Most General Type

- Type inference produces the *most general type*

```
let rec map f lst =
  match lst with
    [] -> []
   | hd :: tl -> f hd :: (map f tl)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Functions may have many less general types

```
val map : (t_1  -> int, [t_1])  -> [int]
val map : (bool -> t_2, [bool]) -> [t_2]
val map : (char -> int, [cChar]) -> [int]
```

- Less general types are all instances of most general type, also called the *principal type*

Complexity of Type Inference Algorithm

- When Hindley/Milner type inference algorithm was developed, its complexity was unknown
- In 1989, Kanellakis, Mairson, and Mitchell proved that the problem was exponential-time complete
- Usually linear in practice though…
  - Running time is exponential in the depth of polymorphic declarations

Type Inference: Key Points

- Type inference computes the types of expressions
  - Does not require type declarations for variables
  - Finds the most general type by solving constraints
  - Leads to polymorphism
- Sometimes better error detection than type checking
  - Type may indicate a programming error even if no type error
- Some costs
  - More difficult to identify program line that causes error
  - Natural implementation requires uniform representation sizes
- Idea can be applied to other program properties
  - Discover properties of program using same kind of analysis

Varieties of Polymorphism

- Parametric polymorphism A single piece of code is typed generically
  - Imperative or first-class polymorphism
  - ML-style or let-polymorphism
- Ad-hoc polymorphism The same expression exhibit different behaviors when viewed in different types
  - Overloading
  - Multi-method dispatch
  - intentional polymorphism
- Subtype polymorphism A single term may have many types using the rule of subsumption allowing to selectively forget information

Summary

- Types are important in modern languages
  - Program organization and documentation
  - Prevent program errors
  - Provide important information to compiler
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types